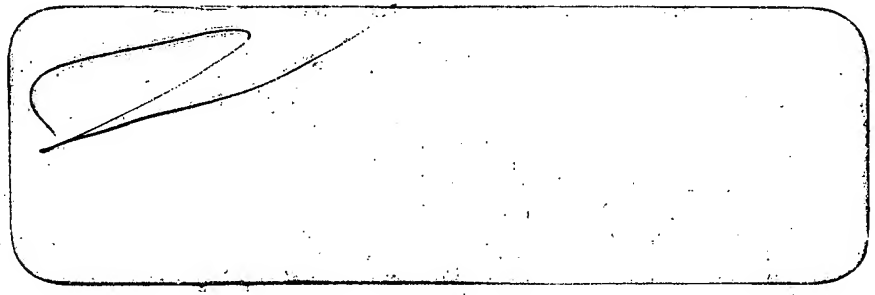
Organization TC2100 Bldg./Rm. RANDOLPH

U. S. DEPARTMENT OF COMMERCE
COMMISSIONER FOR PATENTS
P.O. BOX 1450
ALEXANDRIA, VA 22313-1450
IF UNDELIVERABLE RETURN IN TEN DAYS

OFFICIAL BUSINESS

AN EQUAL OPPORTUNITY EMPLOYER

Intl Priority Airm
U.S. Postage
PAID
Jamaica, NY
Permit No. 9114



Zurück/Retour CN 15

Empfänger/Firma unter der angegebenen Anschrift nicht zu ermitteln	☒ Inconnu/Adresse insuffisante
Empfänger verzogen. Einwilligung zur Weitergabe der neuen Anschrift liegt nicht vor.	☐ Déménagé
Annahme verweigert	☐ Refusé
Nicht abgeholt	☐ Non réclamé
Nicht zulässig	☐ Non admis

Rücksendung am/Retour le: 24/11

RECEIVED
DEC 12 2006
USPTO MAIL CENTER

AIR MAIL

UNITED STATES PATENT AND TRADEMARK C.

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/517,198	12/08/2004	Jean-Paul Theis		3234

7590 10/12/2006

Ante Vistor Gmbh
Harburger Schlossgtrasse 6-12
21079 Hamburg
Germany, 21079
GERMANY

EXAMINER	
GEIB, BENJAMIN P	

ART UNIT	PAPER NUMBER
2181	

DATE MAILED: 10/12/2006

Please find below and/or attached an Office communication concerning this application or proceeding.

Office Action Summary	Application No.	Applicant(s)	
	10/517,198	THEIS, JEAN-PAUL	
	Examiner	Art Unit	
	Benjamin P. Geib	2181	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on *08 December 2004*.

2a)☐ This action is **FINAL**. 2b)☒ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1-13* is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) *1-13* is/are rejected.

7)☐ Claim(s) _____ is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☐ The specification is objected to by the Examiner.

10)☒ The drawing(s) filed on *08 December 2004* is/are: a)☐ accepted or b)☒ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☐ All b)☐ Some * c)☐ None of:

1.☐ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____.

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

FRITZ FLEMING
SUPERVISORY PATENT EXAMINER
TECHNOLOGY CENTER 21
10/4/2006

Attachment(s)

1) ☒ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☐ Information Disclosure Statement(s) (PTO/SB/08) Paper No(s)/Mail Date _____.
4) ☐ Interview Summary (PTO-413) Paper No(s)/Mail Date. _____.
5) ☐ Notice of Informal Patent Application
6) ☐ Other: _____.

DETAILED ACTION

1. Claims 1-13 have been examined.

2. It is hereby acknowledged that the following papers have been received and placed of record in the file: Application on 12/08/2004.

Drawings

3. The drawings are objected to under 37 CFR 1.83(a). The drawings must show every feature of the invention specified in the claims. Therefore, the four method steps (i.e. steps a-d) in claim 1 must be shown or the feature(s) canceled from the claim(s). The drawings show two steps of the method, whereas claim 1 claims four steps. No new matter should be entered.

Corrected drawing sheets in compliance with 37 CFR 1.121(d) are required in reply to the Office action to avoid abandonment of the application. Any amended replacement drawing sheet should include all of the figures appearing on the immediate prior version of the sheet, even if only one figure is being amended. The figure or figure number of an amended drawing should not be labeled as "amended." If a drawing figure is to be canceled, the appropriate figure must be removed from the replacement sheet, and where necessary, the remaining figures must be renumbered and appropriate changes made to the brief description of the several views of the drawings for consistency. Additional replacement sheets may be necessary to show the renumbering of the remaining figures. Each drawing sheet submitted after the filing date of an application must be labeled in the top margin as either "Replacement Sheet" or "New

Sheet" pursuant to 37 CFR 1.121(d). If the changes are not accepted by the examiner, the applicant will be notified and informed of any required corrective action in the next Office action. The objection to the drawings will not be held in abeyance.

Claim Objections

4. Claims 1-12 are objected to because of the following informalities: The phrases "in the following, said dedicated memory is referred to by the term heap address cache" and "said data are also called link data in the following;" in steps b and c, respectively, of claim 1 should be removed. Appropriate correction is required.

5. All claims objected to that have not been specifically addressed above are objected to on the basis of dependence.

Claim Rejections - 35 USC § 102

6. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (b) the invention was patented or described in a printed publication in this or a foreign country or in public use or on sale in this country, more than one year prior to the date of application for patent in the United States.

7. Claims 1-13 are rejected under 35 U.S.C. 102(b) as being anticipated by Lange et al., U.S. Patent No. 6,151,670 (Herein referred to as Lange).

8. Referring to claim 1, Lange has taught a method for implementing autonomous load/store within a data processing system by using symbolic machine code, where said

data processing system comprises a microprocessor and a memory system, where said memory system has a memory hierarchy containing: one or more register files of said microprocessor one or more data caches at different memory hierarchy levels a main memory where said microprocessor has an instruction set and where said instruction set contains one or more instructions of which one or more operands and/or results may specify one or more symbolic variables, where a symbolic machine code is running on said microprocessor, where said symbolic machine code contains one or more instructions of which one or more operands and/or results specify one or more symbolic variables, where said autonomous load/store refers to the loading and storing of data generated and used by said symbolic machine code and where at least part of said data loading and storing is done without requiring any explicit load/store instructions in said symbolic machine code, where each of said symbolic variables specifies one or more entries of a memory other than a register file of said microprocessor, where said entries are used by the microprocessor in order to determine the addresses within the memory hierarchy where the values of said symbolic variables may be stored to and/or loaded from during execution of said symbolic machine code *[column 2, lines 30-50]*, where said method comprises the following steps:

a. when the microprocessor fetches an instruction of which a result and/or one or more operands specify one or more symbolic variables, it computes the addresses within the memory hierarchy where the values of said symbolic variables may be stored into and/or loaded from *[the address of a short or long term register is computed based upon the register identifier; column 2, lines 51-62]*;

b. after anyone of said addresses has been computed, the microprocessor writes this computed address into an entry of a dedicated memory *[register addresses are written to memory with an indication of whether they are short or long term registers; column 3, lines 4-25]*;

c. in addition to said computed address of step b., said microprocessor writes data associated to said computed address into said heap address cache and/or into another memory; said link data are such that, when they are accessed by the microprocessor, they allow the microprocessor to make the link with or to associate them to said computed address and may be used to determine whether said computed address refers to the value of an operand and/or of a result of said instruction *[column 3, lines 4-25]*

d. the microprocessor uses said entry within said heap address cache in order to determine or estimate the lifetime *[term]* of the value to be stored to or to be loaded from said computed address *[column 2, lines 51-62]*

9. Referring to claim 2, Lange has taught a method as claimed in claim 1, where step d. is further specified as follows: the microprocessor uses said entry within said heap address cache and/or said link data in order to determine or estimate the lifetime of said value by determining or estimating the amount of time which elapsed since a previous write of the same address into an entry of said heap address cache *[column 2, lines 40-50]*;

10. Referring to claim 3, Lange has taught a method as claimed in claim 2, where said heap address cache is realized as a circular stack, where steps b. and c. are further specified as follows:

b. after anyone of said addresses has been computed, the microprocessor writes this computed address into the same entry of the circular stack as the one where the link data in step d. are written *[column 2, lines 51-62]*;

c. said microprocessor writes said link data into the same entry of the circular stack as said computed address, said link data comprising one or both of the following: a type-flag, which tells the microprocessor whether the value stored or to be stored at said computed address refers to the value of an instruction operand or of an instruction result a valid-flag, which tells the microprocessor whether the entry contains data which can be overwritten or not *[column 2, lines 51-62]*.

11. Referring to claim 4, Lange has taught a method as claimed in claim 3, where, in addition to the data mentioned in step c., said link data further contain an execution state value, this value allowing to determine the execution state of said symbolic machine code at the point in time when said instruction is fetched or when said link data are written *[column 3, lines 4-25]*.

12. Referring to claim 5, Lange has taught a method as claimed in claim 3, where step b. is further specified as follows: the microprocessor uses said entry within said heap address cache and/or said link data in order to determine or estimate the lifetime of said value by subtracting the entry containing said computed address from another entry of said heap address cache containing the same address *[column 2, lines 51-62]*;

13. Referring to claim 6, Lange has taught a method as claimed in claim 4, where step b. is further specified as follows: the microprocessor uses said entry within said heap address cache and said execution state value in order to determine or estimate the lifetime of said value by subtracting the execution state value stored in the entry containing said computed address from the execution state value stored in the same or in another entry of said heap address cache containing the same address *[column 2, lines 51-62]*;

14. Referring to claim 7, Lange has taught a method as claimed in claim 1, where the microprocessor uses the lifetimes of the values of said symbolic variables in order to determine the addresses and/or the hierarchy levels within the memory hierarchy where said values shall be stored *[column 2, lines 30-50]*;

15. Referring to claim 8, Lange has taught a method as claimed in claim 2, where the microprocessor uses the lifetimes of the values of said symbolic variables in order to determine the addresses and/or the hierarchy levels within the memory hierarchy where said values shall be stored *[column 2, lines 30-50]*;

16. Referring to claim 9, Lange has taught a method as claimed in claim 3, where the microprocessor uses the lifetimes of the values of said symbolic variables in order to determine the addresses and/or the hierarchy levels within the memory hierarchy where said values shall be stored *[column 2, lines 30-50]*;

17. Referring to claim 10, Lange has taught a method as claimed in claim 4, where the microprocessor uses the lifetimes of the values of said symbolic variables in order to

determine the addresses and/or the hierarchy levels within the memory hierarchy where said values shall be stored *[column 2, lines 30-50]*;

18. Referring to claim 11, Lange has taught a method as claimed in claim 5, where the microprocessor uses the lifetimes of the values of said symbolic variables in order to determine the addresses and/or the hierarchy levels within the memory hierarchy where said values shall be stored *[column 2, lines 30-50]* ;

19. Referring to claim 12, Lange has taught a method as claimed in claim 6, where the microprocessor uses the lifetimes of the values of said symbolic variables in order to determine the addresses and/or the hierarchy levels within the memory hierarchy where said values shall be stored *[column 2, lines 30-50]*;

20. Referring to claim 13, Lange has taught a microprocessor having an instruction set containing: one or more instructions of which one or more operands and/or results may specify one or more symbolic variables *[short or long term register identifiers]* one or more symbolic link instructions where said microprocessor is able to execute symbolic machine code *[instructions that include short or long term register identifiers]*, where said symbolic machine code contains one or more instructions of which one or more operands and/or results specify one or more symbolic variables *[column 2, lines 30-50]*.

Conclusion

21. The following is text cited from 37 CFR 1.111(c): In amending in reply to a rejection of claims in an application or patent under reexamination, the applicant or patent owner must clearly point out the patentable novelty which he or she thinks the claims present in view of the state of the art disclosed by the references cited or the objections made. The applicant or patent owner must also show how the amendments avoid such references or objections.

22. The prior art made of record and not relied upon is considered pertinent to applicant's disclosure.

Lozano et al., "Exploiting Short-Lived Variables in Superscalar Processors", teaches storing short-lived variables to memory outside of a processors register file.

Any inquiry concerning this communication or earlier communications from the examiner should be directed to Benjamin P. Geib whose telephone number is (571) 272-8628. The examiner can normally be reached on Mon-Fri 8:30am-5:00pm.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Fritz Fleming can be reached on (571) 272-4145. The fax phone number for the organization where this application or proceeding is assigned is 703-872-9306.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).

Benjamin P Geib
Examiner
Art Unit 2181

FRITZ FLEMING
SUPERVISORY PATENT EXAMINER
TECHNOLOGY CENTER 2100
10/2/2006

Notice of References Cited	Application/Control No. 10/517,198	Applicant(s)/Patent Under Reexamination THEIS, JEAN-PAUL	
	Examiner Benjamin P. Geib	Art Unit 2181	Page 1 of 1

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
*	A	US-6,151,670	11-2000	Lange et al.	712/228
	B	US-			
	C	US-			
	D	US-			
	E	US-			
	F	US-			
	G	US-			
	H	US-			
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	"Exploiting Short-Lived Variables in Superscalar Processors"; Lozano et al.; 1995; IEEE
	V	
	W	
	X	

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

Exploiting Short-Lived Variables in Superscalar Processors

Luis A. Lozano C.* and Guang R. Gao
School of Computer Science
McGill University
Montreal, Quebec, Canada H3A 2A7
lozano@acaps.cs.mcgill.ca, gao@cs.mcgill.ca

Abstract

In this paper, we present experimental evidence showing that a significant number of program variables are short-lived in the sense that their live ranges span only a few instructions. In dynamically scheduled superscalar processors using mechanisms like the reorder buffer, the live ranges for these short-lived variables may occur entirely within the reorder buffer. Therefore, there should be no need to retire (commit) the values produced by these live ranges to the register file. On the basis of this observation, we have proposed a scheme that includes a compiler analysis and a simple architecture extension to avoid the useless commits of the values generated for these short-lived variables. Moreover, we have proposed an extension to the existing register allocation mechanism that does not assign these short-lived variables to locations in the register file. Analyses and results are presented.

1 Introduction

In a superscalar processor, instruction-level parallelism is exploited using a combination of aggressive techniques such as dynamic scheduling [17], speculative execution and register renaming [11]. Dynamic scheduling enables the machine to find multiple independent instructions to be issued and executed each cycle. Speculative execution allows the processor to execute instructions beyond unresolved branches thus increasing the probability of finding more independent instructions. Under register renaming, additional registers are dynamically allocated for each new value generated thus avoiding the anti and output dependencies. The out-of-order instruction processing and the speculative execution lead to the difficult problem of saving a consistent in-order state of sequential execution [10]. Elaborate hardware mechanisms have been devised to support these techniques, including the *reorder buffer* [13, 10] and the *completion buffer* [16].

In the design of superscalar microprocessors, the mechanisms described above require the investment of a substantial portion of transistor budget and chip area. However, the microarchitecture of these mechanisms is not reflected in the instruction-set architecture (ISA) and has remained hidden from the compiler. Although these features provide powerful potential to support instruction parallelism dynamically, it remains a challenge to find a way in which the architects and compiler writers can work together to fully exploit them. We believe that it is important that aggressive features for dynamic instruction scheduling be uncovered to the compiler for possible optimizations.

In this paper, we present an important observation for superscalar processors with the aforementioned mechanisms: a significant number of program variables are "short-lived" in the sense that their whole live ranges occur entirely within the reorder buffer. Therefore, the values of these short-lived variables do not need to be written (committed) back to the register file. We present experimental evidence which demonstrates that a significant portion of the values — often more than 90% — generated in programs are produced by short-lived variables. In the current superscalar implementation, however, the values of these variables are committed to the register file once they leave the reorder buffer. This implies that an unnecessary amount of hardware resources are used during the commit process. Furthermore, since the values of these short-lived variables are never obtained from the register file, why should we allocate them to locations in the register file in the first place?

Based on these observations, we propose a solution which includes: (1) a compile-time analysis method to identify the short-lived variables, (2) a simple architecture extension to avoid the useless commit of the values produced by short-lived variables, and (3) a register allocation scheme by which the compiler can avoid assigning the short-lived variables to the physical registers.

We have organized this paper as follows: In Section 2, we describe the superscalar processor execution model to be used in this paper. Section 3 describes in more detail our observation about the useless commit of short-lived values. Section 4 describes the mechanism used to reduce

*Currently affiliated with Hewlett-Packard, California Language Laboratory. This work was developed as part of the author's M.Sc. Thesis.

1072-4451/95 $4.00 © 1995 IEEE
Proceedings of MICRO-28

the number of useless commits. In Section 5, we describe our scheme for the allocation of the short-lived variables. Section 6 presents the experimental results. In Section 7, we briefly summarize the works from other authors that are related to our research. Finally, in Section 8, we summarize our achievements.

2 The Superscalar Processor Execution Model

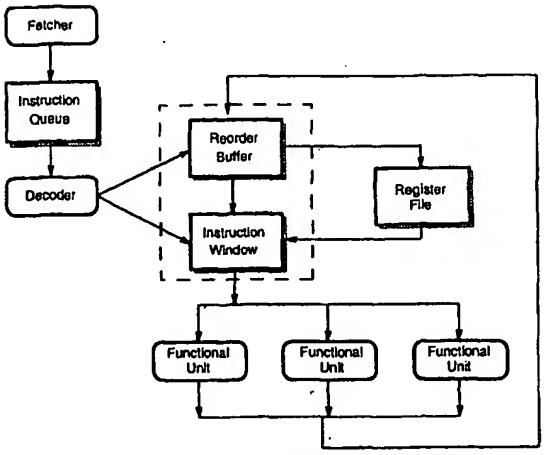


Figure 1: Superscalar processor execution model

Figure 1 shows the structure of a generic superscalar processor model that will be adopted as our execution model. This model has two important components: the instruction window and the reorder buffer. The instruction window serves as a pool of instructions from where the instructions that are ready to execute are issued to the functional units. The reorder buffer is a FIFO structure which ensures that instructions modify the register file in program order and provides the mechanism to support speculative execution and precise interrupts. Although in Figure 1 the reorder buffer and the instruction window are drawn as two separate elements, they could also be assumed to be joined, as is the case for the Register Update Unit [14].

A complete description of this model can be found in [10, 12]. However, let us emphasize some aspects that are important for the understanding of this paper. Instructions enter the reorder buffer in program order at decode time, and their operands are searched for first in the reorder buffer and then in the register file. The destination registers are renamed at decode time by dynamically assigning them a tag that identifies them while they reside in the reorder buffer. Instructions are executed out-of-order and the values calculated are written back to the reorder buffer. These values are not committed to the register file until the corresponding instructions reach the head of the reorder buffer. In order to support speculative execution, instructions that come after a mispredicted branch are marked so that they can be discarded once they reach the head of the buffer.

3 Short-Lived Variables and Useless Commits

3.1 Useless Commits: A Motivating Example

One important observation of this paper is that a significant portion of the values generated by the functional units are used only while they reside inside the reorder buffer. As an example, consider the small loop extracted from the *Tomcatv* benchmark and reproduced in Figure 2(a). Figure 2(b) shows the DLX [7] assembly code for the body of the loop produced with the *dlxcc* compiler, Figure 2(c) shows the live ranges for the different values generated and the respective register assignments. In this figure, small circles (o) denote definition points while cross signs (×) denote the points where the values die. Overlapped cross and circle signs (⊗) denote redefinition points. It can be seen that, for all live ranges, once a new value is defined, it is last used a few instructions later. If we count the length of a live range by the number of instructions between the producer and the last consumer, we can see that the length of the longest live range in this example is 14 instructions. If this code were to be run on a machine with a reorder buffer of 16 entries, all values produced could be consumed while they reside in the reorder buffer. If this were the case, the commit of the instructions to the register file would be useless because none of the values would ever be obtained from the register file.

Another important point is that nine[1] physical registers have been used for the allocation of the temporary variables used in the body of the loop. This is a waste of register names because the values stored in these registers are never being acquired from the register file.

From the analysis of this example, two questions emerge: can we avoid the useless commits of instructions to the register file? Also, can we improve the register allocation process so that register names are not assigned to values that do not require them?

3.2 Experimental Observations

We modified our simulation testbed to establish the percentage of useless commits found during the execution of a set of benchmarks. In Table 1, we present the percentage of useless commits detected when running each benchmark with different sizes of the reorder buffer. A description of the benchmarks used is given in Section 6.

It can be seen that a large percentage of the instructions committed to the register file are useless. The percentage

[1] `f4` and `f6` are being used as double registers.

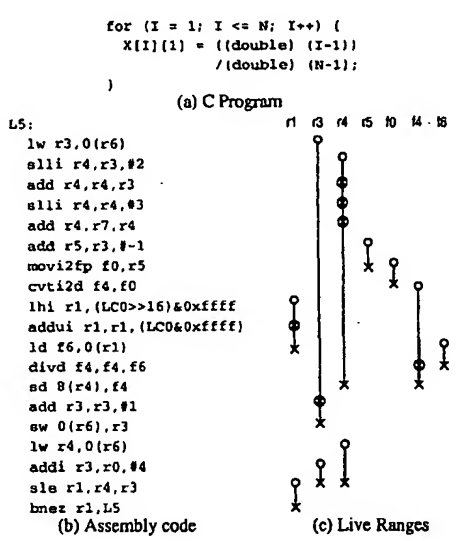


Figure 2: Examples of short-live-range variables

of useless commits increases with the size of the reorder buffer because there is a bigger chance that the last use of a value and its definition point reside in the reorder buffer at the same time. Even with a reorder buffer of 8 entries the percentage of useless commits is very high (on average 89.56%). With a reorder buffer of 32 entries the percentages are even higher, on average 95.11%. These results suggest that all these useless commits constitute a significant waste of resources and allow for architecture and compiler optimizations.

3.3 Problem Formulation

Let $d_0, d_1, d_2 \ldots d_n$ be the definition points for a set of values $v_0, v_1, v_2 \ldots v_n$. Let $u_{i_0}, u_{i_1}, u_{i_2} \ldots u_{i_m}$ be the set of last-use points for a value v_i. We define the live range r_{i_j} of a value v_i as the sequence of instructions in the interval $[d_i, u_{i_j}]$, and the length of r_{i_j}, $\mathcal{L}(r_{i_j})$, as the number of machine instructions in the longest path between d_i and u_{i_j} including d_i and u_{i_j}. Assume we have a superscalar machine $\mathcal{M}$, like the one described in our execution model (Section 2), with a reorder buffer of length $\mathcal{LR}$ entries. We say that the commit of the value v_i produced by the definition point d_i to the register file is a *useless commit* if all the references to v_i are obtained from the reorder buffer rather than from the register file. We say a live range r_{i_j} is short if it has a length that is smaller than or equal to the size of the reorder buffer, i.e., $\mathcal{L}(r_{i_j}) \leq \mathcal{LR}$. Consequently, we call a variable x a *short-lived variable* if all the live ranges associated to x are short. Note that for an instruction at d_k to be a useless commit, it is necessary that *all* the associated live ranges, r_{k_x} for any x, be short.

Benchmark	Buffer Size		
	8	16	32
Alvinn	85.80	88.90	89.89
Bubble	93.82	95.93	97.14
L8	89.66	98.81	98.81
L14	91.65	95.93	97.08
L8unroll	89.54	98.70	98.70
L14unroll	91.77	95.87	97.12
Linpack	92.32	92.61	93.33
Quickrand	81.81	87.52	88.19
Tomcat	95.95	97.61	97.78
Whetstone	83.29	90.21	93.04
Average	89.56	94.21	95.11

Table 1: Percentage of useless commits

Given these definitions, the problems to be studied in this paper can be stated as follows:

- **Problem 1 - Reduction of Useless Commits:** Given a machine $\mathcal{M}$ with a reorder buffer of $\mathcal{LR}$ entries and a program $\mathcal{P}$, we want to develop architecture mechanisms and compiler techniques to reduce the number of useless commits in $\mathcal{M}$, while executing $\mathcal{P}$.
- **Problem 2 - Allocation of short-lived variables:** In conjunction with the solution for Problem 1, we want to propose a register allocation scheme which ensures that the short-lived variables in $\mathcal{P}$ do not occupy physical registers of the machine whenever possible.

Solving Problem 1 will allow the architecture to reduce the number of write ports to the register file without affecting the execution time of the program. The reduction of write ports is an important issue because it simplifies the structure of the data path of the processor [18, 10]. The complexity of implementation of the reorder buffer, the register file and the associated busses depends on the number of ports to the register file. The area complexity of a multiported register file is roughly proportional to the square of the number of ports [3]. In fact, due to the complexity of the register file, the cycle length for many processors is determined by the access time to the register file.

Solving Problem 2 will allow the compiler to utilize more efficiently the physical registers by using them exclusively for the allocation of variables with long live ranges. The effective utilization of the registers is a very important issue since any optimization that tries to increase ILP will also increase register pressure. It is necessary to find mechanisms to reduce the register pressure created by these optimizations and utilize as efficiently as possible the register names provided.

3.4 Solution Strategy

To solve Problem 1 we propose to provide the architecture with specific information so that, at commit time,

it can decide whether a value should be committed to the register file or just discarded. This information should be collected by the compiler by analyzing the characteristics of the live ranges of the variables, and provided to the architecture through the instruction set. Note that only the compiler can detect whether the use of a value is the last one or not. Thus, it is the responsibility of the compiler to flag the instructions that are going to be discarded at commit time.

To solve Problem 2, we propose to modify the register allocation process, so that variables whose live ranges are not committed to the register file are not assigned to physical registers. We maintain that these variables should be assigned to and stored in the reorder buffer instead of the register file. Then, the reorder buffer will be considered an extension of the register file in which the values of short live ranges will be temporarily stored before they are discarded at commit time.

3.5 Issues

To be able to exploit the occurrence of the short-lived variables, we need to solve 3 main issues: 1) the compiler's lack of knowledge about which instructions are present in the reorder buffer at any given point, 2) the detection of useless commits in the presence of the speculative execution of instructions, and 3) the need to provide support for precise interrupts.

The first problem is that, with the current implementation of the reorder buffer, an instruction can be committed to the register file as soon as it reaches the head of the reorder buffer without waiting for the last use of the value to enter the buffer. The second problem is that after a definition point has entered the reorder buffer, one or several branches can be predicted before the last use of the value enters the reorder buffer. Yet, even after the last use has entered, it is not safe to say that the definition point can be discarded once it arrives to the head of the reorder buffer because some of the branches in the middle could have been mispredicted. We will discuss the solutions to these issues in Sections 4.1 and 4.2.

The last issue is related to the need to provide precise interrupts [15, 9]. The problem resides in the recoverability of the processor state after handling the interrupt. If we discard the value produced by the definition point of a short-lived variable, and an instruction lying between the definition point and the last use point of its live range causes an interrupt, then the last use of the value will not be able to find this value after the interrupt has been handled. In order to solve this problem, we propose that at the moment of handling the interrupt, not only the state of the register file and the program counter should be saved, but also the state of the reorder buffer should be saved. In this way we guarantee that the last uses of discarded variables will not have to search for the values in the register file since the forwarding mechanism put them in the corresponding entries in the reorder buffer before the interrupt occurred. The process of saving the state of the reorder buffer can be expensive given that the size of the reorder buffer can be large. However, since interrupts do not occur frequently (around 1 interrupt every 5000 instructions [9]), the performance penalty of saving the state of the reorder buffer should be very small. Also, as we will see in Section 4.2, this mechanism can be further simplified.

4 Reduction of Useless Commits

4.1 Detecting Useless Commits

As suggested in our solution strategy in Section 3.4, the compiler collects information about the live ranges and communicates this information to the processor through the instruction set. Hence, the compiler decides which values should be discarded at commit time and tags these values. In addition, the architecture provides a mechanism to ensure that the whole live range of a tagged value will occur inside the reorder buffer. This can be implemented using two different schemes:

4.1.1 Scheme 1: Marking the last use of a value

In this scheme, the compiler is responsible for informing the architecture which of all the uses of a value is the last one. When the last use enters the reorder buffer an associative search is performed in the reorder buffer and the definition point is marked so that when it arrives at the head of the reorder buffer it can be discarded. This should not increase the time required to enter an instruction into the reorder buffer since the mechanism defined in the execution model is already performing an associative search to find the instruction that is defining the value.

This mechanism is exemplified in Figure 3. This figure shows a fragment of assembly code that has been annotated by the compiler with ampersand signs to indicate which values can be discarded at compile time and to flag the corresponding last uses of these values. Thus, for example, the value of `r3` defined in instruction (2) can be discarded at compile time and the last use of this value is found in instruction (3). The value of `r3` in instruction (3) can be discarded at commit time and its last use is found in instruction (5). On the other hand, the values of `r5` in instruction (1) and of `r4` in instruction (4) must be committed to the register file.

```
(1)  add r5,r0,r3
(2)  slli &r3,r4,#2
(3)  add &r3,&r3,r4
(4)  slli r4,r3,#3
(5)  add &r6,r4,&r3
```

Figure 3: Marking the last use of a value

Special care has to be taken by the compiler so that the use of this mechanism does not produce stalls in the decode stage. If the length of the reorder buffer is $\mathcal{LR}$ and the decode bandwidth of the processor is $\mathcal{DB}$, then the length of the live ranges chosen by the compiler must not exceed $\mathcal{LR} - \mathcal{DB}$. In this way, we avoid the case where instructions cannot enter the reorder buffer because another instruction is at the head waiting for the last use to enter the buffer and stopping the commit of other instructions.

4.1.2 Scheme 2: Keeping a minimum number of instructions in the reorder buffer

In this scheme, an instruction cannot be committed unless there is a minimum number of instructions in the reorder buffer. In this way, the compiler can be sure that if two instructions are separated by a number of instructions that is less than the minimum number of instructions kept in the reorder buffer, the two instructions are going to be simultaneously present in the reorder buffer. Here again, special care has to be taken when choosing the minimum number of instructions to be kept in the reorder buffer in order to avoid decode stalls. The minimum number of instructions $\mathcal{MLR}$ must be set to $(\mathcal{LR} - \mathcal{DB})$. That is, $\mathcal{MLR}$ will be an architectural parameter which depends on the reorder buffer size $\mathcal{LR}$ and the decode bandwidth $\mathcal{DB}$. Knowing $\mathcal{MLR}$, the compiler will be able to decide which live ranges will completely occur inside the reorder buffer and mark them, so the architecture can safely discard the associated value at commit time.

```
(1)  add r5,r0,r3
(2)  slli &r3,r4,#2
(3)  add &r3,r3,r4
(4)  slli r4,r3,#3
(5)  add &r6,r4,r3
(6)  addi &r1,r5,#8
```

Figure 4: Keeping a minimum number of instructions in the reorder buffer.

An advantage of this mechanism is that the compiler does not need to flag the last use of a value to be discarded. Only the definition point has to be flagged. This is illustrated in Figure 4 in which a fragment of assembly code annotated with the information of which values should be discarded at commit time is shown. As it can be seen, the compiler only needs to mark the values that can be discarded at commit time. In this example, the values produced by instructions (2), (3), (5) and (6) can be discarded instead of committed to the register file.

These two schemes are equivalent in terms of effectiveness. The selection of the mechanism will depend, then, on the complexity of the implementation. As mentioned before, Scheme 2 has the advantage that only the definition points need to be marked by the compiler. This implies a smaller change in the instruction format and, depending on the instruction set, could imply that object code compatibility could be preserved. On the other hand, the implementation of Scheme 2 could be more difficult in some machines particularly in the presence of branch prediction.

4.2 Dealing with Instruction Speculation

As we showed in Section 3.5, the problem of detecting useless commits at compile time is more difficult when we consider the speculative execution of instructions. The problem is that it is not safe to discard a definition point even if the last use is also present in the reorder buffer because the last use could be speculatively executing.

After considering a difficult hardware solution to this problem, we decided that the best way to get around this problem was to simply avoid it. We decided to consider only short-lived variables whose definition and last use points are found in the same basic block. The intuition behind this idea is that, since the live ranges we are considering are short, there is a high probability that their definition and last use points are in the same basic block. In general, many of these short live ranges are produced as a consequence of the use of temporary variables by the compiler. Most of the time, these variables introduced by the compiler are used only inside a basic block. In order to verify this observation, we modified our simulator to count the number of useless commits that are caused by live ranges that do not cross the basic block boundaries. The results of our measurements for different reorder buffer sizes are presented in Table 2.

Benchmark	Buffer Size		
	8	16	32
Alvinn	85.80	89.35	89.35
Bubble	93.83	95.90	95.92
L8	89.66	98.81	98.81
L14	91.26	95.16	95.16
L8unroll	89.54	98.70	98.70
L14unroll	91.39	94.85	94.85
Linpack	92.31	92.50	92.64
Quickrand	81.37	84.11	84.50
Tomcat	96.31	97.45	97.57
Whetstone	82.82	89.26	89.26
Average	89.43	93.61	93.68

Table 2: Percentage of useless commits produced by live ranges that do not cross basic block boundaries

Comparing these results with the ones presented in Table 1, it can be seen that most of the short live ranges do not cross the basic block boundaries. Even for a reorder buffer of 32 entries, the difference of the measurements presented in Tables 1 and 2 for a particular benchmark is at most 4%. The differences are further reduced when we consider the measurements for reorder buffers of 8 and 16 entries. This small difference confirms that this scheme can be used,

without a significant loss of precision, to detect the useless commits, thus avoiding in a simple way the problem posed by the speculative execution of instructions.

Another advantage of the use of this scheme is related to the recoverability of the interrupts. As we explained in Section 3.5, in order to make an interrupt recoverable under our scheme, it is necessary to save the state of the whole reorder buffer. An improvement can be obtained if we consider that the short live ranges to be discarded do not cross the basic block boundaries. In this case, the values discarded will only be required by the instructions between the faulting instruction and the next instruction in the reorder buffer that changes the control flow, i.e., by the instructions that are in the same block of the instruction that produced the fault. Thus, it is not necessary to save the status of the whole reorder buffer, but only the status of the entries from the head of the reorder buffer to the next jump or branch instruction. Since jump and branch instructions constitute around 13% of the instruction mix [7], we will be, in general, saving less than eight entries of the reorder buffer each time an interrupt occurs.

4.3 Compiler Analysis

Now that we have selected the hardware mechanism and have simplified the problem in such a way that we can exactly map short live ranges to useless commits, we need to design the compiler analysis to find out which live ranges are going to be tagged so that the values produced are discarded at commit time.

This analysis, which we call the *short-live-range analysis*, must perform the following tasks: 1) Find the length of the longest live range of each variable; 2) Detect variables whose live ranges cross basic block boundaries; 3) Based on the previous information, on the size of the reorder buffer $\mathcal{CR}$, and on the processor's decode bandwidth $\mathcal{DB}$, find the variables that correspond to our definition of a short-lived variable.

The analysis is carried out on the LAST (Low level Abstract Syntax Tree) representation of the McCAT compiler [6]. It performs a backwards walk over the tree keeping track of the definition and last-use points of each live range and the distance between these points. After the analysis is performed we obtain, for every variable in the routine, the length of its longest live range. With this information, and the knowledge of the length of the reorder buffer $\mathcal{CR}$ and the decode bandwidth $\mathcal{DB}$, the analysis can decide which variables are short-lived and pass this information to the code generator. The compiler can produce the code with the flagged live ranges so that the values generated by these live ranges can be discarded at commit time.

5 Allocation of Short-Lived Variables

Up to now, we have only discussed how to modify the architecture so that the values produced by short-lived variables are discarded at commit time. However, a traditional register allocator would continue to assign these variables to physical registers even though the associated values will never be committed to the register file. Hence, our idea is to expose the reorder buffer to the compiler as an extension of the programmable registers for storing short-lived values. We propose to use the space in the reorder buffer as additional registers which we call *symbolic registers*. There are as many symbolic registers as entries in the reorder buffer. However, a symbolic register is not tied to any particular location in the reorder buffer. For example, if a value is assigned to the symbolic register two (`sr2`), that does not mean the value will be stored in the second entry of the reorder buffer. The actual association between a symbolic register and a reorder buffer entry is done at runtime. When the instruction defining the value of the symbolic register enters the reorder buffer, the renaming mechanism renames the symbolic register and assigns an entry in the reorder buffer to store the value being produced.

The problem now is to modify the compiler to make an efficient use of the symbolic registers. To achieve this, we propose a register allocation scheme that is performed in four steps: 1) Short-live-range analysis; 2) Allocation of short-lived variables; 3) Modified Chaitin-like allocation for remaining variables; and 4) Introduction of spill code. These steps are depicted in Figure 5. As it can be seen, we are adding two steps to the traditional Chaitin-like allocation process [4, 2] to allocate the short-lived variables first. As shown in the figure, the whole allocation process has to be repeated after the introduction of spill code.

In the following subsections, we will explain, in more detail, the steps involved in the proposed allocation scheme.

5.1 Allocation of Short-Lived Variables

The first step is to perform the short-live-range analysis, as we described in Section 4.3, to find out which variables can be allocated to the symbolic registers. The second step is to assign the symbolic registers to these short-lived variables. Since the live ranges of these variables do not cross the basic block boundaries, they can be allocated in linear time using a simplified version of the algorithm for register allocation at the basic block level presented in [1].

Using this algorithm, independent live ranges that belong to the same variable can be allocated to different registers. Also, since the number of symbolic registers available is equal to the number of entries in the reorder buffer, and since the length of each live range does not exceed the length of the reorder buffer (the short-live-range analysis ensures this is true), the number of symbolic registers is always enough to allocate all the short-lived variables without requiring the introduction of spill code.

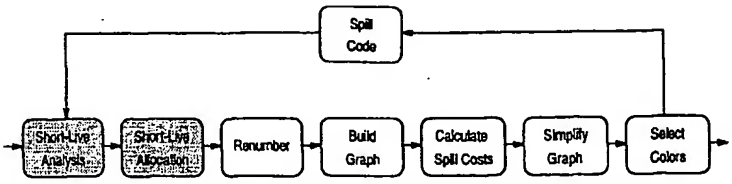


Figure 5: Modified register allocation for short-lived variables

In Figure 6, we give an example of how the code looks when the variables are allocated using the traditional Chaitin-like allocation scheme (Figure 6(b)), compared to the code produced when we use our proposed allocation scheme (Figure 6(c)). Note that our scheme uses three symbolic registers and two physical registers compared to the five physical registers used by the traditional allocation scheme[2].

```
t1 = &a;          add r6,r30,#-36     add sr1,r30,#-36
t2 = 8 * i;       slli r3,r5,#3       slli sr2,r5,#3
t3 = t2 + 4;      addi r3,r3,#4       addi sr2,sr2,#4
t4 = t1 + t3;     add r5,r6,r3        add sr2,sr1,sr2
t1 = &b;          add r6,r30,#-40     add sr1,r30,#-40
t2 = 8 * j;       slli r3,r4,#3       slli sr3,r4,#3
t3 = t2 + 4;      addi r3,r3,#4       addi sr3,sr3,#4
t5 = t1 + t3;     add r4,r6,r3        add sr1,sr1,sr3
c  = *t4;         lw r8,0(r5)         lw r4,0(sr2)
d  = *t5;         lw r3,0(r4)         lw r5,0(sr1)
```

(a) Program fragment | (b) Code produced after Chaitin-like allocation | (c) After separate allocation for short-lived variables

Figure 6: An example of the code produced for the two allocation schemes

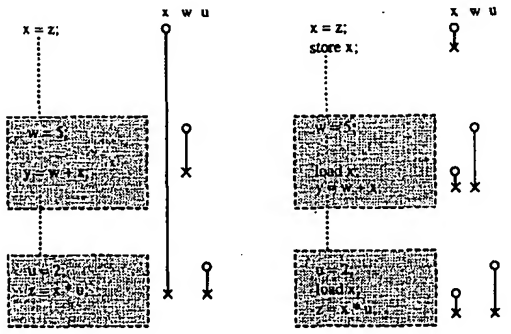
5.2 Allocation of the Long-Lived Variables

After all the short-lived variables are allocated, the next step is to allocate the remaining variables using Chaitin's allocator with the Briggs improvement [4, 2]. The advantage of our scheme at this point is that, since many of the variables have already been allocated in the previous step, the size of the interference graph is smaller than what it would be for the traditional allocator, thus simplifying the problem. Therefore, fewer variables are competing for the physical registers which means a decrease in the amount of spill code required. Yet, it is still possible that during the selection of the colors for the interference graph, the Chaitin allocator decides to spill some variables to memory thus requiring the introduction of spill code.

5.3 Introduction of Spill Code

When introducing spill code, the register allocator inserts a load before each use and a store after each definition for each variable being spilled. The introduction of these load/store instructions has two effects on our mechanism: 1) The variables that are spilled now become short-lived variables. 2) The introduction of loads and stores may cause some short-lived variables to become long-lived.

These two effects can be better understood by examining the example presented in Figure 7. Figure 7(a) presents a fragment of a program together with the representation of the corresponding live ranges before the spill code is introduced. The gray boxes represent zones where the register pressure is assumed to be high enough to require the introduction of spill code[3]. In the example, x is not short-lived and has several uses, while w and u are short-lived variables. For the purposes of our example, we are going to assume that the length of the live range of w is the maximum length for w to be considered short-lived. Let us assume that, since the register pressure is high, the register allocator selects x to be spilled, and let us examine the effect of introducing spill code for this variable, as depicted in Figure 7(b). The first effect is that since x was spilled, its live range is now composed of several small live ranges causing x to become a short-lived variable that can be allocated using symbolic registers. This is an advantage of our scheme over Chaitin's allocator which repeats the whole allocation process again because spilled variables continue to interfere with the other variables in the interference graph.



(a) Before spill code (b) After spill code

Figure 7: Effect of spill code on the allocation of short-lived variables

The second effect of the introduction of spill code is that

[2]We do not count r30 here since in DLX this register is reserved to store the value of the frame pointer

[3]We assume that some registers are occupied by other program variables not shown in the figure.

since the length of the live range for w was on the limit of being considered short, and since some instructions were introduced for the spill code of x, then the live range for w is not short anymore, and w has to be allocated using the physical registers. This forces the repetition of the whole process to check if anymore spill code is required for the variables that interfere with w. We have observed, though, that this does not happen too often and that our process converges faster than in the traditional Chaitin allocation. Finally, note that despite the fact that the live range for u is longer now, u is still a short-lived variable and can still be allocated using symbolic registers. This implies that no additional spill code is going to be introduced in this zone with high register pressure.

6 Experimental Results

All analyses and experiments were carried out using the McCAT testbed [6, 12]. For our experiments, we have used 10 benchmarks which are briefly described in Table 3. We included some integer kernels taken from non scientific applications, some floating point kernels from scientific code and some complete floating point scientific applications. We believe that although the sample of benchmarks is small, it is representative of a wide set of applications.

Benchmark	Description
Alvinn	Neural network to control an autonomous vehicle - SPEC92
Bubble	Recursive bubble sort
L8	Loop 8 of the Livermoore loops
L14	Loop 14 of the Livermoore loops
L8unroll	Loop 8 unrolled twice
L14unroll	Loop 14 unrolled twice
Linpack	Kernel routines from the "LINPACK" linear algebra package
Quickrand	Recursive quicksort
Tomcat	Mesh generation - SPEC 92
Whetstone	Floating point intensive synthetic benchmark

Table 3: Benchmarks

6.1 Base Model

All experiments were performed by simulating the behavior of the execution model described in Section 2. We instantiated this model by specifying the values of the different parameters as shown in Table 4. The processor resulting from this specification is our *base model* for the experiments to be carried out. Some of the parameters of the base model, like the decode bandwidth, the number of memory accesses per cycle and the memory latency, were selected according to real values found in some recent superscalar processor designs. The base model uses a reorder buffer size of 16 entries and assumes 4 write ports per register file. The different measurements are obtained by varying some parameters of the base model, e.g., the reorder buffer size between 8, 16 and 32 entries, and the number of register write ports per register file. In such cases, the sizes of the instruction dispatch window and the load and store buffers are always kept equal to the size of the reorder buffer. For simplicity, we assume that the base model and its variations have a perfect cache, that we have enough functional units (five of each type) to execute the operations that are not related to memory accesses and that the latency of these operations is one cycle.

Parameter	Value
Instruction queue	16 instructions
Decode bandwidth	4 instr. per cycle
Reorder buffer	16 entries
Memory latency	2 cycles
Memory accesses	1 per cycle
Address resolution functional units	1
Other functional units	5 of each type
Latency of other functional units	1 cycle
Write ports per register file	4

Table 4: Configuration parameters for the base model

6.2 Summary of Results

The major results of our experiments are:

- The short-live-range analysis can be successfully used to avoid the useless commit of instructions to the register files. The proposed analysis captures most of the short-lived variables: on the average, close to 90% of all variables are detected to be short-lived when we assume a reorder buffer with 16 entries, and more than 90% when the reorder buffer size is increased to 32. The combined architecture and compiler scheme can eliminate the majority of the useless writes to the register files: on the average 87% for the base model and close to 90% when reorder buffer size is increased to 32.
- The mechanism devised to avoid the useless commits of instructions can be used to reduce the number of write ports to the register files without affecting performance. In fact, using this mechanism we could reduce the number of write ports down to one and obtain a loss on performance of only one percent.
- The proposed method for allocation of short-lived variables can decrease the amount of spill code needed thus improving execution time. When the register pressure is high (only 4 registers are effectively available) the improvement exceeds 22% for a reorder

buffer of size 16, and 26% when the reorder buffer size is increased to 32.

We will further elaborate on these results in the following three sections.

6.3 The Effect of Short-live-range Analysis and Architecture Support for Useless Commit Elimination

The effectiveness of the short-live-range compiler analysis (explained in Section 4.3) and of the hardware mechanisms that allow the elimination of the useless writes to the register file (explained in Section 4.1) is illustrated in Table 5. In this table are the measurements of the effectiveness of our combined hardware/software mechanism for different sizes of the reorder buffer. For each size, we present the percentage of variables that were detected by the compiler to be short-lived variables, and the percentage of writes that were discarded at run time. It can be seen, that even for a small reorder buffer (of size 8), on the average more than 80% of the variables used at the low level representation of the program are detected as short-lived variables. This is caused, as explained before, by the large number of temporary variables introduced by the compiler. Most of these variables, plus the ones that are spilled to memory by the register allocator, are successfully captured by our compiler analysis: on the average close to 90% (89.34) when the reorder buffer size is 16 (base model), and 80.13% and 91.92% when the reorder buffer sizes are 8 and 32 respectively.

Furthermore, the proposed architecture mechanism can make use of the information provided by the compiler and eliminate a great majority of useless writes to the register files. The reduction on the average is 88% (87.98) for the base model, and is 76.34% and 89.71% when the reorder buffer sizes are 8 and 32 respectively. It can be seen that in some cases the percentage of discarded writes is very high, more than 98%. On the other hand, the lowest values are still considerably high. For the base model, the lowest percentage is 76%. Moreover, if we compare the results of this table to the results presented in Table 1, we can see that our analysis is able to detect a great majority of the useless commits. For reorder buffers of sizes 16 and 32, the difference is less than 7%. For a reorder buffer with 8 entries, the difference is less than 14%.

6.4 The Effect of Reducing the Number of Ports to the Register File

Since the percentage of commits that can be discarded is large, we decided to measure the loss of performance of the proposed (optimized) model when the number of write ports per register file is restricted, and compare it to the loss of performance for the base model under the same restriction. The results of the comparisons are tabulated in Table 6. The left side of Table 6 shows the relative performance obtained by varying the number of register write ports when the compiler and architecture optimization proposed in this paper is not applied. We report the performance of variations of the base model when the number of write ports to the register files is restricted to 1, 2 or 3 ports per register file compared (in normalized form) to the performance of the base model when the number of ports is 4. It can be seen that the restrictions on the number of ports to the register file can seriously affect the performance of the base model. Using, for example, only one port per register file can degrade the performance of the base model by 55%. If we increase the number of ports to two per register file we still degrade performance by 18%.

We applied the proposed optimization to the base model and performed the same measurements. The results, on the right side of Table 6, show that the loss in performance is very small, as expected. Around 1% when using only one write port per register file.

6.5 Effect of the Allocation of Short-Lived Variables

Table 7 shows the improvement obtained in the overall execution time of the benchmarks by using the short-lived variables allocation method explained in Section 5. To show the effectiveness of the method, we vary the number of registers available for the register allocation process (4, 8 and 16), and the size of the reorder buffer (8, 16 and 32) from the base model. The performance improvement is calculated with the formula *(cycles(traditional) - cycles(proposed))/cycles(traditional)*. Where *traditional* refers to the Chaitin like allocation method and *proposed* refers to our improved allocation scheme.

It can be seen that the higher the register pressure, the higher the improvement obtained by our method. When the register pressure is high (only 4 registers are available), the improvement is significant: over 22% (22.35%) for a reorder buffer of size 16, and 15.96% and 26.43% when the reorder buffer size is 8 and 32 respectively. It can also be seen that for a given level of register pressure (number of registers available) the improvement that can be obtained depends on the size of the reorder buffer. The reason for this is that with larger reorder buffers there are better chances to find more short-lived variables and, therefore, further reduce the register pressure.

It is important to note that all the measurements assume a perfect cache and that we use large load/store buffers. The improvements obtained by our method would be even greater without the use of these features.

7 Related Work

In [14], Pleszkun and Sohi remark upon the occurrence of the useless commit of instructions to the register file

Benchmark	Reorder Buffer					
	8		16		32	
	Short Variables	Discarded Writes	Short Variables	Discarded Writes	Short Variables	Discarded Writes
Alvinn	75.33	69.38	84.44	88.13	86.44	88.47
Bubble	81.25	79.56	89.58	89.76	91.67	91.81
L8	73.15	72.04	89.49	96.06	94.16	98.80
L14	83.33	83.95	88.33	88.37	91.67	90.78
L8unroll	81.37	70.90	95.24	95.60	96.70	98.34
L14unroll	89.35	83.97	92.73	88.41	94.29	90.80
Linpack	80.20	84.37	89.93	90.88	94.14	91.46
Quickrand	76.60	71.73	80.85	76.61	80.85	76.61
Tomcat	79.05	80.26	91.38	88.17	94.80	89.99
Whetstone	81.62	67.27	91.39	77.76	94.49	80.00
Average	80.13	76.34	89.34	87.98	91.92	89.71

Table 5: Effectiveness of the short-live-range analysis and the architecture support for reducing useless commits

Benchmark	Base Model			Optimized		
	Write Ports			Write Ports		
	1	2	3	1	2	3
Alvinn	0.46	0.82	0.97	1.00	1.00	1.00
Bubble	0.38	0.74	0.94	1.00	1.00	1.00
L8	0.34	0.69	0.97	1.00	1.00	1.00
L14	0.44	0.85	1.00	0.96	1.00	1.00
L8unroll	0.34	0.68	0.96	1.00	1.00	1.00
L14unroll	0.44	0.87	1.00	0.96	1.00	1.00
Linpack	0.45	0.86	1.00	1.00	1.00	1.00
Quickrand	0.66	0.95	1.00	1.00	1.00	1.00
Tomcat	0.43	0.80	0.99	0.99	1.00	1.00
Whetstone	0.55	0.96	1.00	0.99	1.00	1.00
Average	0.45	0.82	0.98	0.99	1.00	1.00

Table 6: Performance effect of the number of write ports

when using state maintenance mechanisms like the reorder buffer or the Register Update Unit – RUU. Based on this, they suggest that the number of write ports to the register file could be reduced without affecting performance. However, they do not perform any experiments on these observations.

In [13], Pleszkun et al. propose exposing the structure of the reorder buffer to the compiler to improve the performance of the processor. In this way, their compiler is able to improve the scheduling of the instructions and provide speculative execution. The authors also propose to keep the number of instructions in the reorder buffer constant and to handle the interruptions by saving the state of the reorder buffer. Our work has been influenced by theirs, but our objectives are different. Their work does not make explicit use of the renaming capabilities of the reorder buffer.

In [5], Franklin and Sohi make an analysis of the characteristics of the communication between instructions through registers. They conclude that many of the values generated are used only once and that most of the values are dead soon after they have been created, i.e., between 30-40 instructions later. They also observe that, by using a mechanism to buffer 30 or more instructions, at least 80% of the writes to the register file are unnecessary. Our experimental results reported in Section 3.2 and Section 6 were derived independently. The observation that most of the short-lived variables have live ranges within a basic block is new. Also, our solution strategy is novel and can effectively work in the presence of speculative execution.

Finally, in [8], Hoogerbrugge and Corporaal study the register file port requirements of Transport Triggered Architectures (TTA). The researchers show they can eliminate 65% of the write traffic to the register file by forwarding values directly between functional units. Our scheme can eliminate a larger percentage of writes to the register file because of the buffering of instructions in the reorder buffer. This buffering allows the forwarding of values between instructions that are not as close together as it is required in the TTA forwarding mechanism.

We are not aware of any research that explicitly shows how to modify the register allocator so as not to assign short-lived variables to physical registers.

8 Conclusions

In this paper we have seen that by allowing the compiler to access the different resources used by the architecture, the implementation of some features in hardware can be simplified and a better utilization of the provided resources can be obtained. In particular, we have demonstrated how compiler and architecture techniques can be combined to take advantage of the fact that many program variables are short-lived. Our implementation and experimental results have provided evidence that the proposed optimizations can be effective and may lead to significant performance improvements with relatively few architectural modifications.

Benchmark	Number of registers								
	4			8			16		
	Reorder Buffer			Reorder Buffer			Reorder Buffer		
	8	16	32	8	16	32	8	16	32
Alvinn	21.56	16.81	15.44	0.33	0.22	0.60	0.00	0.01	0.01
Bubble	11.00	19.42	30.52	0.04	0.06	0.04	0.04	0.06	0.04
L8	15.50	28.24	31.97	5.18	6.48	6.52	0.03	0.05	0.07
L14	14.83	20.51	25.65	3.31	0.11	6.84	2.18	2.16	2.72
L8unroll	25.68	36.57	42.01	5.27	9.11	11.23	0.65	1.63	2.00
L14unroll	20.60	27.76	26.70	5.89	9.34	16.57	6.43	6.30	4.88
Linpack	12.92	15.30	27.08	0.15	1.80	1.11	0.70	1.58	1.67
Quickrand	15.89	26.94	32.50	0.23	3.59	5.56	0.00	2.26	4.17
Tomcat	11.81	16.15	14.69	3.35	8.32	7.63	1.73	3.41	4.07
Whetstone	9.76	15.83	17.73	5.27	13.10	14.29	7.64	16.20	15.88
Average	15.96	22.35	26.43	2.90	5.21	7.04	1.94	3.37	3.55

Table 7: Percentage of improvement obtained by allocation of short-lived variables

Acknowledgments

We would like to thank Claudia Pateras for the great help provided in writing this paper. The authors are thankful to the anonymous reviewers for their constructive comments and to the members of the ACAPS group for the useful discussions we had with them. We also acknowledge Hewlett-Packard for its support. This work was supported by a research grant from the Natural Science and Engineering Research Council (NSERC), Canada.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Pub. Co., Reading, Mass., corrected edition, 1988.

[2] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice U., Houston, Tex., Apr. 1992. Publ. as Rice COMP TR92-183.

[3] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proc. of MICRO-25*, pages 292–300, Portland, Ore., Dec. 1992.

[4] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, Jan. 1981.

[5] Manoj Franklin and Gurindar S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proc. of MICRO-25*, pages 236–245, Portland, Ore., Dec. 1992.

[6] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proc. of the 5th Intl. Work. on Languages and Compilers for Parallel Computing*, number 757 in LNCS, pages 406–420, New Haven, Conn., Aug. 1992. Springer-Verlag. Publ. in 1993.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Pub., Inc., 1990.

[8] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of transport triggered architectures. In *Proc. of MICRO-27*, pages 191–195, San Jose, Calif., Nov.–Dec. 1994.

[9] Wen-mei W. Hwu and Yale N. Patt. Checkpoint Repair for Out-of-order Execution Machines. In *Proc. of ISCA-14*, pages 18–26, Pittsburgh, Penn., Jun. 1987.

[10] Mike Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Inc., Englewood Cliffs, N. Jersey, 1991.

[11] Robert M. Keller. Look-ahead processors. *ACM Computing Surveys*, 7(4):177–195, Dec. 1975.

[12] Cecile Moura. SuperDLX – a generic superscalar simulator. ACAPS Tech. Memo 64, Sch. of Comp. Sci., McGill U., Montréal, Qué., Apr. 1993.

[13] A. R. Pleszkun, J. R. Goodman, W-C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. B. Schechter. WISQ: A restartable architecture using queues. In *Proc. of ISCA-14*, pages 290–299, Pittsburgh, Penn., Jun. 1987.

[14] A. R. Pleszkun and G. S. Sohi. The performance potential of multiple functional unit processors. In *Proc. of ISCA-15*, pages 37–44, Honolulu, Hawaii, May–Jun. 1988.

[15] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. on Computers*, 37(5):562–573, May 1988.

[16] Peter Song and Marvin Denman. *The PowerPC 604 RISC Microprocessor*. Motorola; IBM Corporation, 1994.

[17] J. E. Thornton. Parallel operation in the Control Data 6600. In *AFIPS Fall Joint Computer Conf.*, pages 33–40, 1964.

[18] Augustus K. Uht and Darin B. Johnson. Data path issues in a highly concurrent machine. In *Proc. of MICRO-25*, pages 115–118, Portland, Ore., Dec. 1992.